Chapter 14 OOP

Introduction

What is OOP? OOP is short for Object Oriented Programming and implies an object is the focus of the program. This chapter discusses the use of objects in PLC programming and their use for making programs that are more readable. Software engineering in general has looked for programming tools that allow more robust and reliable computer programs.

Earlier programming discussions of “top-down” computer programming and “structured programming” have evolved into the use of objects that encapsulate an idea and stand alone in the program as that evolved concept. The overall idea of any programming endeavour has been to break the problem down into component parts and solve each part as it pertains to the whole and then combine the parts into a unified overall program. While the idea of a data structure is important, the focus of a top-down program or structured program would focus on the structure or flow rather than the task.

OOP

The idea of pluggable entities or “lego” modules implies that modules can be created that can be plugged into one another and perform as a whole. This entails the idea of information that may be needed only inside the “lego” program and thus hidden from the outside world, in short, the idea of encapsulation. This leads to the idea of objects and object oriented programming or OOP. In OOP, the data is protected since it can be manipulated only inside the shell or protected program called the object. The logic is protected since it is only executed inside the object and the details of the program are known inside the object as to how the program and data interact.

OOP protects the data. It is easier to write in a general sense since the module can block out other programming considerations and the data can be stored inside the OOP as opposed to elsewhere. Data can, obviously, be stored anywhere the programmer wants but the protection of the OOP gives a security that wasn’t available in earlier programming methods. And the programmer and the end user can focus on the object. This gives a great advantage to maintaining the program since the focus is placed where it was originally intended, on the object itself. And if some part changes, the programmer is reminded of the entity that is being affected and that changes should be studied as a whole for the object and not for just a part of the object. In the case of maintainability of control programs for the factory floor, the diagnostic part should be modified when the control portion is modified. If encapsulated, the programmer is reminded of both together since both should be encapsulated inside the object.

The programming language C refers to an object as a class. Other languages have similar names. In Siemens S7-Basic, the object is referred to as a Function (FC) or Function Block (FB). Allen-Bradley’s RSLogix 5000 also has introduced a function with version 18 called the AOI or add-on instruction. While it will not be featured in this chapter, its use is similar to the function or function block described here.

To program an FC or FB, first identify the object or objects involved. Identify the messages or...
signals the object needs to respond to and the outputs that result from these messages or signals. The FC or FB, while considered a class, may also be considered a template for the program. And the idea of FB’s or FC’s calling other FB’s or FC’s is a powerful concept and creates the idea of sub classes or sub-sub classes. The FC or FB can also be re-used again and again in the same or other programs for the same or different clients. This gives the idea of a class or OOP a huge advantage over conventional programming as we build programs over the years or from job to job.

Arguments for the benefits of reusability include:

- Reliability
- Efficiency of programming effort
- Saving time in program development
- Decreased maintenance effort
- Resulting cost savings
- Consistency of programs

Some additional terms of OOP include:

- Encapsulation – combining of programs and data to manipulate outcome in an object
- Inheritance – building a hierarch of objects, each with inheritance of the parent object
- Polymorphism – allowing one set of actions to share an object with another set of actions

STEP7 allows the user to create containers of user created program code. This helps to logically structure the program, speeds programming, and shortens the commissioning time by using proven code over and over. These containers are called Blocks. Blocks are added to the project with SIMATIC Manager and are programmed in the same fashion as an OB. Once the blocks are programmed and debugged, they can be called as needed to implement the control function. Functions (FCs) are the first type of container we will use.

FCs can be locked by the creator. This helps to preserve and protect the code and can actually help to simplify the overall program by firmly defining a functionality that is unchanged from one instance of code to another. STEP7 enables the user to create a storage location for custom functions called a Library. Several frequently used standardized and system functions are provided to the user in several libraries included with STEP7. The user can create a custom library and add items as needed, supporting programming standardization across projects.

The S7 architecture also supports the structuring of user-defined data storage locations, called data blocks, and reusable data templates called PLC Data Types.

The S7-1200 and 1500 controllers use programming elements that comply with IEC 61131-3 standard. At the core of the programming structure are code and data containers, known collectively as “Blocks”. The programmable logic controller provides various types of blocks in which the user program and the related data can be stored. Depending on the requirements of the process, the program can be structured in different blocks.
Organization blocks: (OB’s) form the interface between the operating system and the user program. The entire program can be stored in OB1 that is cyclically called by the operating system (linear program) or the program can be divided and stored in several blocks (structured program).

Function: (FC’s) contains a partial functionality of the program. It is possible to program functions so that they can be assigned parameters. As a result, functions are also suited for programming recurring, complex partial functionalities such as calculations. System functions (SFC) are parameter-assignable functions integrated in the CPU’s operating system. Both their number and their functionality are fixed. More information can be found in the Online Help.

Function Block: (FB’s) offer the same possibilities as functions. In addition, function blocks have their own memory area in the form of instance data blocks. As a result, function blocks are suited for programming frequently recurring, complex functionalities such as closed-loop control tasks. System function blocks (SFB) are parameter-assignable functions integrated in the CPU’s operating system. Both their number and their functionality are fixed.

Data Blocks: (DB’s) are data areas of the user program in which user data is managed in a structured manner.

Permissible Operations: You can use the entire operation set in all blocks (FB, FC and OB). We will now start to explore these basic “program structuring elements” of S7 beginning with the FC, or Function. A Function is defined by the IEC 61131 standard as a code container that does not retain internal values from one scan to the next. Functions in the S7 PLC behave in this fashion, and act as a container for user developed program code. A function may have a set of local variables defined for use within the function. Typically, when “called” in the main program, a function will have new “values” (or actual parameters) loaded into the local variable (called formal parameters) for use during execution of the function. Once the “results” are calculated and function execution finishes, the resulting “output value(s)” get returned to the main program. The “Add” instruction shown here is a good example of a “function.” In this example the value 125 is assigned to variable “A” and 16 is assigned to variable “B”, the code is then executed (add A to B), and the resulting answer (141) is returned. The add function can be used again, with different values passed into A and B.

Application:

You can program parameter-assignable blocks for frequently recurring program functions. This has the following advantages:

• The program only has to be created once, which significantly reduces programming time.
• The block is only stored in the user memory once, which significantly reduces the amount of memory used.
• The block or the functionality implemented with the block can be called as often as you like, each time with a different address. For this, the formal parameters (input, output, or in/out parameters) are supplied with different actual operands every time they are called. You cannot assign parameters to organization blocks since they are called directly by the operating system. Since no programmed block call takes place in the user program, it is not possible to pass actual operands to an OB.
Before you can create the program for the parameter-assignable FC, you have to define the formal parameters in the declaration table. In the table in the slide, you can see the three different parameter types that are available for use in the block. It is up to the programmer to select the declaration type for each formal parameter.

- The ‘IN‘ declaration type should be assigned only to declaration types that will be read for instructions in the subroutine.
- Use the ‘OUT‘ declaration type for parameters that will be written to within the function.
- Use the “IN_OUT” for formal parameters that need to have a reading access and a writing access, such as a bit passed into the block that is used in the block for an edge operation.
- TEMP variables are intended to be used for holding interim calculation values or other values that are only required when the block is executing. TEMP variables exist in the local data stack while the block is executing and are overwritten when the block exits. The TEMP variables are - even though they are listed under "Interface" - not components of the block interface, since they do not become visible when the block is called and that no actual parameters have to be passed for the declared TEMP variables in the calling block.

The interface of a block forms the IN, OUT, and IN_OUT parameters. The RETURN parameter is a defined, additional OUT parameter that has a specific name according to IEC 61131-3. This parameter only exists in FCs in the interface. The declared formal parameters of a block are its interface to the "outside" meaning they are "visible" or relevant to other blocks that call this block. If the interface of a block is changed by deleting or adding formal parameters later on, then the calls have to be updated.

When an FC is added to a project, the FC is accessible via the Project Browser. When the FC is to be executed must be determined. This is defined by which OB in which the FC is to be called. For example, if the FC is to be executed every scan, it is placed in OB1. To call an FC in OB1, drag and drop the FC from the project browser onto a network.

**Blocks Types**

This is primarily a Siemens concept although Allen-Bradley has also introduced the idea with their function blocks in later versions of RSLogix 5000.

**Global DBs**

A data block (DB) is a data area in the user program containing user data. Global data blocks store data that can be used by all other blocks. The structure of the global data blocks is user defined.
Several Types of Blocks in STEP 7 Basic

Interaction between the operating system and the various block types is pictured below:

The operating system calls this program cycle OB once in each cycle and thereby starts the execution of the user program. The main program code is executed in this OB. You can have more than one. OB 1 will be the first with additional cyclic OBs starting at 200 and above.

When the CPU’s operating mode changes from stop to run a "startup" event occurs. You can have more than one "startup OB. OB 100 will be the first with additional startup OBs starting at 200 and above.

For time-delay interrupts, the operating system starts the corresponding "time-delay interrupt" OBs after a user defined delay time. The delay time starts running after the call of the instruction SRT_DINT. Together with the "cyclic interrupts" there is a limit to four of these OB types. For Cyclic Interrupts, these interrupt OBs serve to start program code execution in periodic time intervals independently of the cyclic program execution. Together with the "time-delay interrupts" there is a limit to four of these OB types.
These interrupt OBs can be triggered by high-speed counters and input channels.

If a diagnostics-capable module in which the diagnostic error interrupt is enabled detects an error, this module triggers a diagnostic error interrupt. There is only one OB with the fixed number 82.

The operating system calls the time error interrupt, OB 80, if one of the following events occurs:
1. The cyclic program exceeds the maximum cycle time
2. The called OB is currently being executed
3. An overflow has occurred in an interrupt OB queue
4. Interrupt loss due to high interrupt load
Creating a Data Block

Follow the steps below to create a Data Block:

1. Double-click the "Add new block" command
2. The "Add new block" dialog box opens
3. Click the "Data block (DB)" button
4. Enter a name for the data block
5. Select the "Manual" option button if you want to assign the number of the block
6. If you choose to use manual assignment, enter the block number in the input field.
7. To add additional properties for the data new block, click the arrow beside "Further information" in the lower part of the dialog box. An area with further input fields is displayed.
8. Enter all the properties you require.
9. Confirm your entry with "OK".

Creating an FC Block

FC - Function:
1. Code blocks without memory
2. For the programming of often needed complex functions
3. After the function has been executed, the data in the temporary tags is lost
4. For storing data permanently, functions have to use data blocks

Follow the steps below:

1. Double-click the "Add new block" command.
2. The "Add new block" dialog box opens.
3. Click the "function (FC)" button.
4. Enter a name for the block.
5. In the "Language" drop-down list, select the programming language for the new block.
6. Select the "Manual" option button if you want to assign the number of the block.
7. If you choose to use manual assignment, enter the block number in the input field.
8. To add additional properties for the new block, click the arrow beside "Further information" in the lower part of the dialog box.
9. An area with further input fields is displayed.
10. Enter all the properties you require.
11. Confirm your entries with "OK".
The Block Interface

Before you can create the program for the parameter-assignable block, you have to define the block parameters in the Interface table. The block interface allows local tags to be created and managed. The tags are subdivided into two groups shown by the table below:

| Block parameters that form the block interface when it is called in the program |
|---|---|---|---|
| Type                   | Section | Function                                      | Available in                                      |
| Input parameters       | Input   | Parameters whose values are read by the block  | Functions, function blocks and some types of organization blocks |
| Output parameters      | Output  | Parameters whose values are written by the block | Functions and function blocks                      |
| InOut Parameters       | InOut   | Parameters whose values are read by the block when it is called, and whose values are written again by the block after execution | Functions and function blocks                      |

| Local data that are used for storage of intermediate results |
|---|---|---|---|
| Type                   | Section | Function                                      | Available in                                      |
| Temporary local data   | Temp    | Tags that are used to store temporary intermediate results. Temporary local data are retained for only one cycle | Functions, function blocks and organization blocks |
| Static local data      | Static  | Tags that are used for storage of static intermediate results in the instance data block. Static data is retained until overwritten, which may be after several cycles | Function blocks only |

Temporary Tags

General
Temporary tags can be used in all FC and FB blocks, and some OB blocks. They are used to temporarily store information while the block is being executed. The data are lost when the block is exited.

Interface Tag
You define the temporary tags in the interface table of the block. In the "TEMP" row you enter a tag name and the associated data type.

Access
At the beginning of a block execution, all temporary tags have an indefinite value. When working with temporary tags, you must therefore
make sure that the tag is first of all assigned a defined value before it is queried.
In the example, the result of subtracting is assigned to the temporary variable "Aux_Result" before it is then queried by the adder.

Note
Tags that begin with the # special character indicate that they are block interface tags. Block interface tags are only valid and usable in the block in which they are declared.
The Program Editor automatically inserts the # character.

Temp Tags
The figure below demonstrates how temp tags only occupy memory while that block is active. For example the values of the temp tags for FC17 can only be accessed from within FC17 and the values will remain as long as FC17 is active. When you reach the end of FC17 and return back to OB1 the temp tags for FC17 will not retain their value.

Creating an FB Block

FB – Function block
Code blocks that store their values permanently in instance data blocks, so that they remain available even after the block has been executed.

All In-, Out-, InOut- parameters are stored in the instance DB – the instance DB is the memory of the FB.

Definition
Function blocks are code blocks that store their values permanently in instance data blocks, so that they remain available even after the block has been executed. They save their input, output and in/out parameters permanently in the instance data blocks. Consequently, the parameters are still available after the block execution. Therefore they are also referred to as blocks "with memory".

Block Interface
The block interface for an FB looks similar to that of an FC. There are two groups of Block interface tags:
1. Block parameters that form the block interface when it is called in the program.
   - Input, Output, and In/Out parameters are a part of this group

2. Local data that are used for storage of intermediate results
   - Temporary local data and Static local data are part of this group

**Static Local Data**

An instance DB is used to save static local data. These static local data can only be used in the FB, in whose block interface table they are declared. When the block is exited, they are retained.

**Parameters**

When the function block is called, the values of the actual parameters are stored in the instance data block.

If no actual parameter is assigned to an interface parameter in a block call, then the last value stored in the instance DB for this parameter is used in the program execution.

You can specify different actual parameters with every FB call. When the function block is exited, the data in the data block is retained. To keep the data unique for each instance of a call it is required to assign a different instance DB each time a call instruction to an FB is written in code.

You can program parameter-assignable blocks for frequently recurring program code. This has the following advantages:

1. The program only has to be created once, which significantly reduces programming time.

2. The block is only stored in the user memory once, which significantly reduces the amount of memory used.

3. The FB can be called as often as you like, each time with a different address assignment. For this, the interface parameters (input, output, or in/out parameters) are supplied with different actual operands every time called.

**Program Execution**

When the instance of the FB1 call is executed, the Interface tag "#Fault" is replaced with the data of the PLC tag "FaultSignal" (I0.7). The value is actually loaded into the instance DB memory location for "#Fault" and the instruction uses that DB memory location for its value. If no parameter was assigned at the interface of the call, then whatever value stored in the DB memory location "#Fault" is used.

**Parameter-Assignability**

Both FC or FB blocks as parameter-assignable, but FC always require parameter assignment since they do not have their own memory (instance DB).
Reusability

Even if the "Pilot Light Control" is required twice in the system, you only have to program FB1 once with its block interface assignment.

The FB1 is then called twice for the two different faults and is assigned a different actual address each time. Each call will require a different instance DB.

Multi-instance data block

Definition

Multi-instances enable a called function block to store its data in the instance data block of the calling function block. This allows you to concentrate the instance data in one instance data block and thus make better use of the number of instance data blocks available.

When to use

You can only call up function blocks as multiple instances if they are included in the libraries supplied with STEP 7 V11.2, such as timers and counters. You cannot call up any function blocks you have created yourself as multiple instances.

Example Task: Controlling a Press – Configuring an FC

Our first program consists of programming a press control.

A press with a protective guard is to be activated with a START button S3 only if the protective screen is closed. This state is monitored with a sensor Protective Screen Closed B1.

If this is the case, a 5/2 way valve M0 for the press cylinder is set so that a plastic shape can be pressed.

The press is to retract again when the EMERGENCY OFF button (break contact) EMERGENCY OFF is activated, or the sensor Protective Screen B1 no longer responds, or the sensor Cylinder B2 Extended responds.

Assignment list:

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%I 0.1</td>
<td>EMERGENCY OFF</td>
<td>EMERGENCY OFF button (break contact)</td>
</tr>
<tr>
<td>%I 0.3</td>
<td>S3</td>
<td>Start button S3 (make contact)</td>
</tr>
<tr>
<td>%I 0.4</td>
<td>B1</td>
<td>Sensor protective screen closed (make contact)</td>
</tr>
<tr>
<td>%I 0.5</td>
<td>B2</td>
<td>Sensor Cylinder extended (make contact)</td>
</tr>
<tr>
<td>%Q 0.0</td>
<td>M0</td>
<td>Extend Cylinder A</td>
</tr>
</tbody>
</table>
Program structure of the example:

**Organizaton block OB1**

Block called cyclically by the operating system. Here, Function FC1 is used.

**Function FC1**

In this example, contains the actual program for the press control. Is called by OB1.
The contents of the function are created using the network below created as FC1 Press.

First select in project navigation the Controller Press[CPU1214C DC/DC/DC] and then PLC variables. With a double click, open the table PLC variables and as shown below, enter the names for the inputs and outputs (→ Controller Press[CPU1214C DC/DC/DC] → PLC Variables → PLC Variables)
The program sequence is written in so-called blocks. As a matter of standard, organization block OB1 already exists. It represents the interface to the CPU’s operating system, is called by it automatically, and processed cyclically.

From this organization block, additional blocks can be called in turn for structured programming, such as the function FC1. The purpose is to break down an overall task into partial tasks. These can be solved more easily and tested in their functionality.

To generate the function FC1, in Project Navigation first select Controller Press[CPU1214C DC/DC/DC] and then Program blocks. Next, double click on Insert new block (→ Controller Press[CPU1214C DC/DC/DC] → Program blocks → Insert new block)

Fig. 14-5   Adding a New FC Block

In the selection, select Function (FC) and assign the name Program press. As programming language, FBD is entered. Enumeration is automatic. Since this FC1 is called later with the symbolic name anyhow, the number is no longer that important. Accept your input with OK. (→ Function (FC) → Program Press → FBD → OK)
The block Program Press[FC1] will be opened automatically. However, before the program can be written, the block’s interface has to be declared. When the interface is declared, the local variables known only in this block are specified.

The variables consist of two groups

- Block parameters that generate the interface of the block for the call in the program.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Function</th>
<th>Available in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input parameters</td>
<td>Input</td>
<td>Parameters whose values the block reads</td>
<td>Functions, function blocks and some types of organization blocks</td>
</tr>
<tr>
<td>Output parameters</td>
<td>Output</td>
<td>Parameters whose values the block writes</td>
<td>Functions and function blocks</td>
</tr>
<tr>
<td>InOut parameters</td>
<td>InOut</td>
<td>Parameters whose value the block reads when it is called and after processing, writes again to the same parameter</td>
<td>Functions and function blocks</td>
</tr>
</tbody>
</table>
Local data that is used for storing intermediate results:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Function</th>
<th>Available in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporary local data</td>
<td>Temp</td>
<td>Variables that are used for storing temporary intermediate results.</td>
<td>Functions, function blocks and organization blocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Temporary data is retained for one cycle only.</td>
<td></td>
</tr>
<tr>
<td>Static local data</td>
<td>Static</td>
<td>Variables that are used for storing static intermediate results in the instance data block. Static data is retained also over several cycles, until it is written anew.</td>
<td>Function blocks</td>
</tr>
</tbody>
</table>

When declaring the local variables, in our example the following variables are needed.

**Input:**  
- **EMERGENCY_OFF**  
  EMERGENCY OFF monitoring is entered  
- **Start**  
  the start button is entered  
- **B_Screen**  
  the status of the protection screen is entered  
- **B_Cylinder**  
  the status of the sensor Cylinder Extended is entered

**Output:**  
- **M_Press**  
  a status for the output Press Cylinder is written

**Temp:**  
- **HM01**  
  Auxiliary flag 01 for the SR FlipFlop

All variables in this case are of the type 'Bool'; which means variables that only can have the status '0' (false) or '1' (true). To make it easier to follow them, all local variables should also be provided with a sufficient comment.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Name</th>
<th>Data type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EMERGENCY_OFF</td>
<td>Bool</td>
<td>Emergency OFF</td>
</tr>
<tr>
<td>3</td>
<td>Start</td>
<td>Bool</td>
<td>pushbutton START</td>
</tr>
<tr>
<td>4</td>
<td>B_safety_fence</td>
<td>Bool</td>
<td>sensor safety fence closed</td>
</tr>
<tr>
<td>5</td>
<td>B_cylinder</td>
<td>Bool</td>
<td>sensor cylinder moved out</td>
</tr>
<tr>
<td>6</td>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>M_Press</td>
<td>Bool</td>
<td>press cylinder</td>
</tr>
<tr>
<td>8</td>
<td>InOut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Temp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>HM01</td>
<td>Bool</td>
<td>temporary memory bit 01</td>
</tr>
<tr>
<td>12</td>
<td>Retem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Ret_val</td>
<td>Void</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 14-7

After having declared the local variables, we can now start programming. To provide a better overview, we program in networks. A new network can be inserted by clicking on the symbol
Insert network. Like the block itself, each network should be documented in the title line. If a longer text is needed for the description, the Comment field can be used (→)

To generate our solution, we need an SR Flipflop. It is located under Instructions in the folder Bit combinations. If you point with the mouse to an object such as the SR flipflop, detail information about this object will be displayed (→ Instructions → Bit combination → SR)

Fig. 14-8  Entering the FC Program

If you highlight an object and then press F1 on your PC, you will be provided with online help about this object in a window to the right (→ F1)

Note: Here, extensive information is provided in online help regarding the function and the wiring of the SR flipflop.

Now, drag the SR flipflop with the mouse below the comment in Network 1. (→ SR)
Next, highlight the Set input of the SR flipflop and click on AND in the favorites (\(\rightarrow S \rightarrow\) Favorites \(\rightarrow\) AND)

Exactly in the same way we place the OR at the R1 input and the assignment at the Q output of the SR flipflop (\(\rightarrow R1 \rightarrow OR \rightarrow Q \rightarrow\) assignment)
Now, we enter the local variables. It suffices to enter the first letter of the local variables in the fields at the commands. Then we can select the desired variable from a list. Local variables are always identified with the symbol '#' preceding the name (→ #M_Press).

Likewise, add the other local variables. At the OR, another input is to be inserted. To this end, highlight the lowest input with the right mouse key and select 'Insert input' (→ Insert input). Assign the local variable shown here to this input also. If an input is to be inverted, simply drag the symbol Negation from the Favorites to the input. (→ Favorites → Negation)
Next, the Properties of the cyclically processed block Main[OB1] are selected. Block properties can be modified (→ Properties → Main[OB1])

In the properties, select the programming-Language function block diagram FBD. (→ FBD → OK)
As mentioned previously, the block "Program Press" has to be called from the program block Main[OB1]. Otherwise, the block would not be processed at all. Open this block by double clicking on Main[OB1] (→ Main[OB1])

![Fig. 14-15](image)

The block Program Press can then simply be dragged with Drag&Drop to Network 1 of the block Main[OB1]. Don’t forget to document the networks also in block Main[OB1] (→ Program Press).

![Fig. 14-16](image)
Next, the interface parameters of the block "Program Press" have to be connected to global PLC variables. It is sufficient to enter the first letter of the global variable in the field before the local variable of the block. Then, the desired operand can be selected from a list (→ "EMERGENCY OFF")

Likewise, connect the input variables Start, B_Screen and B_Cylinder as well as the output variable M_Press to the PLC variables shown here. With a mouse click on Store project the project will be stored. (→ “S3“ → "B1“ → "B2“ → "M0“ → )
To load your entire program into the CPU, first highlight the folder Controller Press and then click on the symbol Load to device (→ Controller Press →) and download.

**Sample Task for Belt Control – Configuring an FB**

When blocks are generated and if they are to work in any program like a ‘Black Box’, they have to be programmed by using variables. In this case, the following rule applies: that in these blocks, no absolute-addressed inputs/outputs, flags etc. must be used. Within the block, only variables and constants are used.

In the example below, a function block is to be generated with a variable declaration that contains a belt control that is dependent on the operating mode.

With button 'S1', the operating mode 'Manual' and with button 'S2' the operating mode 'Automatic' can be selected.

In the operating mode 'Manual', the motor is switched on as long as button 'S3' is operated, whereby button 'S4' must not be operated.

In the operating mode 'Automatic', the belt motor is switched on with button 'S3' and switched off with button 'S4' (break contact).

**Assignment list:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%I 0.0</td>
<td>S1</td>
<td>Button operating mode Manual S1 (make contact)</td>
</tr>
<tr>
<td>%I 0.1</td>
<td>S2</td>
<td>Button operating mode Automatic S2 (make contact)</td>
</tr>
<tr>
<td>%I 0.2</td>
<td>S3</td>
<td>On button S3 (make contact)</td>
</tr>
<tr>
<td>%I 0.3</td>
<td>S4</td>
<td>Off button S4 (break contact)</td>
</tr>
<tr>
<td>%Q 0.2</td>
<td>M01</td>
<td>belt motor M01</td>
</tr>
</tbody>
</table>

**Note:** The off button S4 is a break contact here in order to ensure wire break safety. That means: if there is a wire break at this button, the system stops automatically. Otherwise, it could not be stopped if there were a wire break. For that reason, in control engineering all stop buttons, off buttons/switches have to be designed as break contacts.

Since modern programming is not carried out with absolute addresses but with variables, the global PLC variables have to be specified. These global PLC variables are descriptive names with comments for those inputs and outputs that are used in the program. Later, during programming, this name is used to access the global PLC variables. These global variables can be used in the entire program, in all blocks.

To this end, select in Project Navigation the Controller Belt - CPU1214C DC/DC/DC and then PLC variables. Open the table PLC variables with a double click and enter the names for the inputs and outputs as shown below (→ Controller Belt[CPU1214C DC/DC/DC] → PLC variables→ PLC variables)
To generate the function block FB1, first select Controller Belt[CPU1214C DC/DC/DC] in project navigation, and then Program blocks. Now, double click on Add new block (→ Controller Belt[CPU1214C DC/DC/DC] → Program blocks → Add new block)
In the selection, select Function block (FB) and assign the name Belt. As programming language, we specify function block diagram FBD. Enumeration is automatic. Since this FB1 is called later with the symbolic name anyhow, this number is no longer that important. Accept the input with OK (→ Function block (FB1) → Belt → FBD → OK)

![Add new block](image)

The block Belt[FB1] will be opened automatically. But before we can write the program, we have to declare the block’s interface.

When the interface is declared, the local variables - known only in this block- are specified. The variables are divided into two groups:

- Block parameters that generate the interface of the block for the call in the program.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Function</th>
<th>Available in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input parameters</td>
<td>Input</td>
<td>Parameters whose values the block reads</td>
<td>Functions, function blocks and some types of organization blocks</td>
</tr>
</tbody>
</table>
Output parameters | Output | Parameters whose values the block writes | Functions and function blocks
--- | --- | --- | ---
In/out parameters | InOut | Parameters whose value the block reads when called, and after processing writes to the same parameter | Functions and function blocks

- Local data that is used for storing intermediate results

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Function</th>
<th>Available in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporary local data</td>
<td>Temp</td>
<td>Variables that are used for storing temporary intermediate results. Temporary data is retained for one cycle only.</td>
<td>Functions, function blocks and organization blocks</td>
</tr>
<tr>
<td>Static local data</td>
<td>Static</td>
<td>Variables that are used for storing static intermediate results in instance data blocks. Static data is retained - also over several cycles - until it is written anew.</td>
<td>Function blocks</td>
</tr>
</tbody>
</table>

To declare local variables, the following variables are needed for our example.

**Input:**

- man: the button for selecting the operating mode Manual is entered (make contact)
- auto: the button for selecting the operating mode Automatic is entered (make contact)
- on: the ON button is entered (make contact)
- off: the OFF button is entered (break contact)

**Output:**

- motor: the status for the output Belt Motor is written

**Static (exists only in the function blocks FB):**

- automan: the preselected operating mode is stored
- motorauto: we store when the belt motor was started in the Automatic mode
All variables are of the type 'Bool'; that means, binary variables that only can have the status '0' (false) or '1' (true).

In this example, it is important to note that the status of the two variables 'automan' and 'motorauto' has to be stored over a longer period of time. For that reason, the variable type Static has to be used here. This variable type in turn exists only in a function block FB. For the sake of clarity, all local variables should also be provided with a sufficient comment.

After the local variables have been declared, the program can now be entered by using the variable names (variables are identified with the symbol '#'). For the example in FBD, it could look like this:
Next, the Properties of the cyclically processed block Main[OB1] are selected. Block properties can be changed ( → Properties → Main[OB1])

In the properties, select the programming Language function block diagram FBD ( → FBD → OK)
Now, the block "Belt" has to be called from the program block Main[OB1]. Otherwise, the block would not be processed. With a double click on Main[OB1], open this block (→ Main[OB1])

![Fig. 14-26]

You now can drag the block Belt[FB1] with Drag&Drop to Network 1 of the block Main[OB1]. Don’t forget to document the networks also in block Main[OB1] (→ Belt[FB1])

![Fig. 14-27]
Since we are dealing with a function block, it has to be provided with memory. In SIMATIC S7-1200, data blocks are provided as memory. Such an assigned data block is called Instance Data block. Here, it is to be specified and generated automatically (→ automatic→ OK)

![Call options](image1)

In OB1, we now connect the input variables and the output variable with the PLC variables shown here. By clicking on "Save project", the project is stored (→ "S1“ → "S2“ → "S3“ → "S4“ → "M01“ → "Save project")

![Save project](image2)
To load your entire program into the CPU, first highlight the folder Controller Belt and then click on the symbol \( \text{Load to device} \) (\( \rightarrow \) Controller Belt \( \rightarrow \)).

Now, start the CPU by clicking on the symbol \( \text{(a)} \) (\( \rightarrow \)).

By clicking on the symbol \( \text{Monitoring On/Off} \), you can, during the program test, observe the status of the input and output variables at the block “Belt“, but also the program execution in the block Belt (\( \rightarrow \) Belt[FB1] \( \rightarrow \)).
Since our block “Belt” was generated according to the rules for standard blocks (no use of global variables within the block), it can be used and called any number of times. Below, an expanded PLC variable table is shown, with the inputs and outputs for two belts.

### PLC tags

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Datatype</th>
<th>Address</th>
<th>Retain</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1_conv1</td>
<td>Bool</td>
<td>%Q.0</td>
<td></td>
<td>conveyor1 pushbutton manual mode S1 (no contact)</td>
</tr>
<tr>
<td>2</td>
<td>S2_conv1</td>
<td>Bool</td>
<td>%Q.1</td>
<td></td>
<td>conveyor1 pushbutton automatic mode S2 (no contact)</td>
</tr>
<tr>
<td>3</td>
<td>S3_conv1</td>
<td>Bool</td>
<td>%Q.2</td>
<td></td>
<td>conveyor1 pushbutton ON S3 (no contact)</td>
</tr>
<tr>
<td>4</td>
<td>S4_conv1</td>
<td>Bool</td>
<td>%Q.3</td>
<td></td>
<td>conveyor1 pushbutton OFF S4 (no contact)</td>
</tr>
<tr>
<td>5</td>
<td>N01_conv1</td>
<td>Bool</td>
<td>%QO.2</td>
<td></td>
<td>conveyor1 motor conveyor belt N01</td>
</tr>
<tr>
<td>6</td>
<td>S1_conv2</td>
<td>Bool</td>
<td>%Q.4</td>
<td></td>
<td>conveyor2 pushbutton manual mode S1 (no contact)</td>
</tr>
<tr>
<td>7</td>
<td>S2_conv2</td>
<td>Bool</td>
<td>%Q.5</td>
<td></td>
<td>conveyor2 pushbutton automatic mode S2 (no contact)</td>
</tr>
<tr>
<td>8</td>
<td>S3_conv2</td>
<td>Bool</td>
<td>%Q.6</td>
<td></td>
<td>conveyor2 pushbutton ON S3 (no contact)</td>
</tr>
<tr>
<td>9</td>
<td>S4_conv2</td>
<td>Bool</td>
<td>%Q.7</td>
<td></td>
<td>conveyor2 pushbutton OFF S4 (no contact)</td>
</tr>
<tr>
<td>10</td>
<td>N01_conv2</td>
<td>Bool</td>
<td>%QO.3</td>
<td></td>
<td>conveyor2 motor conveyor belt N01</td>
</tr>
</tbody>
</table>
Now, the block Belt can also be called twice in OB1, with different wiring respectively. For each call, another instance data block is specified.
Lab 14.1

Revisit the Binary Addition/Binary Subtraction lab from chapter 8 to subtract one 16 bit word from another and put the 16 bit result in a third word using a function and using the Siemens TIA software.

Remember:

\[
\begin{align*}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1 \\
1 + 1 &= 0 \text{ carry 1} \\
1 + 0 + \text{carry} &= 0 \text{ carry 1} \\
1 + 1 + \text{carry} &= 1 \text{ carry 1}
\end{align*}
\]

These are the rules for binary addition.

To see binary addition at work:

\[
\begin{array}{c}
\text{Carry} \\
\text{Number 1} \\
\text{+ Number 2} \\
\text{Results}
\end{array}
\begin{array}{c}
1 \ 1 \ 1 \ 1 \\
0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \\
0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\
1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0
\end{array}
\]

Binary addition may take place in ladder logic. Instructions are provided to carry out this function (ADD), but it is worthwhile to examine the process of binary addition using ladder logic.

Since Bit 0 does not have a carry-in, half-adder logic may be employed but only for this bit. It can be seen that half-adder logic is simpler than full-add logic by comparing Fig. 8-35 (Half-Adder) to Fig. 8-36 (Full Adder).
Accessing Bits in Words (Siemens)

Examples

In the PLC tag table, “DW” is a declared tag of type DWORD. The examples show bit, byte, and word slice access:

<table>
<thead>
<tr>
<th></th>
<th>LAD</th>
<th>FBD</th>
<th>SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit access</td>
<td>&quot;DW&quot;.b[11]</td>
<td></td>
<td>IF &quot;DW&quot;.b11 THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>... END_IF;</td>
</tr>
<tr>
<td>Byte access</td>
<td>&quot;DW&quot;.b2 = &quot;DW&quot;.b3</td>
<td></td>
<td>IF &quot;DW&quot;.b2 = &quot;DW&quot;.b3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>... END_IF;</td>
</tr>
<tr>
<td>Word access</td>
<td>AND Word</td>
<td></td>
<td>out: = &quot;DW&quot;.w0 AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;DW&quot;.w1;</td>
</tr>
</tbody>
</table>

Accessing a tag with an AT overlay

The AT tag overlay allows you to access an already-declared tag of a standard access block with an overlaid declaration of a different data type. You can, for example, address the individual bits of a tag of a Byte, Word, or DWord data type with an Array of Bool. To overlay a parameter, declare an additional parameter directly after the parameter that is to be overlaid and select the data type “AT”. The editor creates the overlay, and you can then choose the data type, struct, or array that you wish to use for the overlay.

Example

This example shows the input parameters of a standard-access FB. The byte tag B1 is overlaid with an array of Booleans:

<table>
<thead>
<tr>
<th>B1</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT[0]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[1]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[2]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[3]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[4]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[5]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[6]</td>
<td>Bool</td>
</tr>
<tr>
<td>AT[7]</td>
<td>Bool</td>
</tr>
</tbody>
</table>

Table 4-8 Overlay of a byte with a Boolean array

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Another example is a DWord tag overlaid with a Struct:

<table>
<thead>
<tr>
<th>DWord</th>
<th>DWord_Struct</th>
<th>AT &quot;DWord&quot;</th>
<th>Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT[1]</td>
<td>AT &quot;Word&quot;</td>
<td>AT &quot;Byte&quot;</td>
<td>AT &quot;Byte&quot;</td>
</tr>
</tbody>
</table>
Rules
- Overlaying of tags is only possible in FB and FC blocks with standard access.
- You can overlay parameters for all block types and all declaration sections.
- An overlaid parameter can be used like any other block parameter.
- You cannot overlay parameters of type VARIANT
- The size of the overlaying parameter must be less than or equal to the size of the overlaid parameter.
- The overlaying variable must be declared immediately after the variable that it overlays and identified with the keyword “AT”.

Complete the lab using a function instead of coding each network as separate logic.

**Binary Subtraction:**
To perform binary subtraction, the easiest method is to find the 2’s complement of the second number and then add the two numbers together.

The best method of finding the 2’s complement requires the use of a memory bit. The rule requires that bits from the original number be copied to the 2’s complement number starting at the right-most bit. The rule applies until a “1” is encountered. The first “1” is copied but a memory bit is set after which the bits are “flipped”. Try this rule. It works and may be employed using ladder logic and a Latch bit to quickly find the 2’s complement of a number. The logic for finding the 2’s complement of a number in ladder logic is begun in Fig. 8-37. Again, logic must be added to complete the function using rungs similar to rungs 4 and 5 of this figure but using bits 2 through 15.

Again, code the logic using a function.
Problems

1. Three types of parameters for interface of a function are:
   a
   b
   c

2. Local data is of two types. They are:
   a
   b

3. Data blocks are either Single ____ or multi _____. What is the deciding factor which to use?

4. List some program blocks that are standard.

5. Describe a function or function block that would have the title “Engine” and have two types of engines that could be called – Diesel or Gasoline.

6. What is the A-B process for adding a function?

   For the next three programs, only use skeletal statements but enough to get the idea:

7. In the Kitchen:
   In the kitchen are several needs for automation including cooking breakfast. In the breakfast shelf are several kinds of cereal including oatmeal, cream-of-wheat and grits. Each requires the microwave and a cooking time. Each requires an amount to be weighed on a scale. When the weight is achieved, the bowl is placed in the microwave for a time period. Write a program using FC’s or FB’s to achieve cooking of the breakfast cereal.

8. Three numbers are to be added and the result displayed. Use a FC or FB to accomplish this.

9. In OB1, there is a FC1 accessed that subtracts 1 from a number. Build the function block SUB1 to complete the operation. Show all tables and logic:
10. In OB1, there is a FC1 accessed that does the following:

```
FC1
"INV"

"S3" In_Num

"B0" In_Bit

Out_Num "X3"
```

Copy In_Num into Out_Num except for the bit 0 which is the exclusive OR of Bit 0 and the variable ‘B0’.

Build the function block “INV” to complete the operation. Show all tables and logic:

11. In Siemens’ OB1, there is a FC1 accessed that does the following:

```
FC1
"INV"

"S3" In_Num

"S3By" In_Byt

Out_Num "X3"
```

Copy In_Num into Out_Num except for the bits 0 to 7 which are ‘anded’ with In_Byt

Build the function “INV” to complete the operation. Show all tables and logic inside the Function (FC):

Additional useful information:
12. In Siemens’ OB1, there is a FC1 accessed that does the following:

```
FC1
“MIX”
“S1” In_Num_1
“S2” In_Num_2
“S3” Bit_Num
Out_Num “X3”
```

Bit_Num determines how many bits are moved from In_Num_1. Remaining bits are moved from In_Num_2. For example, if Bit_Num=7, the number of bits moved from In_Num_1 is 7 as shown at left.

Build the function “MIX” to complete the operation. Show all tables and logic inside the Function (FC1):

http://www.youtube.com/watch?v=aUILkF4aI30&feature=relmfu

Siemens SIMATIC S7-1200 Part 2 - Re-Usable Libraries
See how easy it is to implement reusable Libraries in Step 7 Basic Software eliminating time consuming coding of repeat functions. This is part two of a four part series showcasing the time and cost saving benefits of the new S7-1200 and its Step 7 Basic development software. For more information see: http://www.usa.siemens.com/s7-1200

http://www.youtube.com/watch?v=L2NLcAQhiSg&feature=relmfu

Siemens SIMATIC S7-1200 Part 4 - Project-wide Cross Referencing Made Easy
See how easy it is to troubleshoot the complete Controller and HMI software project together for both SIMATIC Basic HMI panels and S7-1200 Controllers. This is part four of a four part series showcasing the time and cost saving benefits of the new S7-1200 and its Step 7 Basic development software. For more information see: http://www.usa.siemens.com/s7-1200